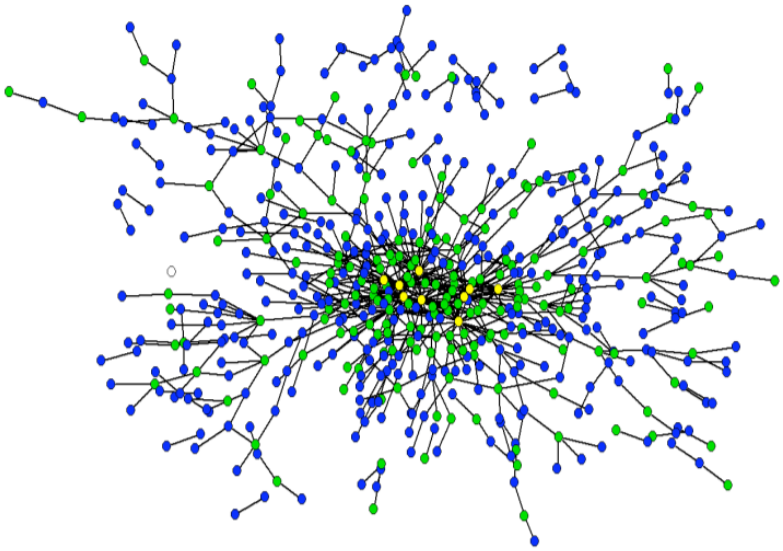
COMP 251 - Network Flows

Algorithms and Data Structures

Written By

David Knapik

McGill University Department of Mathematics and Statistics
david.knapik@mcgill.ca



March 25, 2018

Contents

1 Introduction

Lets say that a good is produced at some source vertex s. We must send as much of the good as possible to a sink vertex t. We have that each arc $a = (i, j)$ has a capacity $u_a = u_{ij}$; the maximum amount of the good that can be sent via that arc (say over a given time period).

Definition 1.0.1. A **network flow** $\vec{f}$ from a source s to a sink t satisfies two properties:

1. Capacity Constraints - the flow on an arc a is non-negative and at most its capacity. I.e. $0 \leq f_a \leq u_a$.
2. Flow Conservation - the flow into a vertex $v \neq \{s, t\}$ equals the flow out of the vertex. I.e. $\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a$

Definition 1.0.2. The **value** of a flow $\vec{f}$ is the quantity of flow that reaches the sink t. I.e.

$$value = |f| = \sum_{a \in \delta^-(t)} f_a = \sum_{a \in \delta^+(s)} f_a$$

So the fundamental problem we consider is to find an $s - t$ flow of maximum value. I.e., we want the $\max_{a \in \delta^-(t)} f_a$ such that $\sum_{a \in \delta^-(v)} f_a = \sum_{a \in \delta^+(v)} f_a$ for all $v \in V \setminus \{s, t\}$ and $0 \leq f_a \leq u_a$ for all $a \in A$.

1.1 The Flow Decomposition Theorem

1.1.1 Paths

What do $s - t$ paths have to do with $s - t$ flows? Well, an $s - t$ path is just a flow of value one. So, the union of a set of $s - t$ paths still satisfies flow conservation. And, if in addition this union of paths satisfies the capacity constraint $f_a \leq u_a$ on each arc, then they form an $s - t$ flow.

1.1.2 Cycles

What do directed cycles have to do with $s-t$ flows? Well, a directed cycle also satisfies flow conservation. Thus, the union of a set of cycles satisfies flow conservation. So, if in addition this unions of cycles satisfies the capacity constraint on each arc, then they form an $s - t$ flow.

1.1.3 Putting it Together

So, from the above, we conclude that the union of a set of paths and a set of cycles satisfies flow conservation. So, if this union also satisfies the capacity constraint on each arc, then we have an $s - t$ flow. We note that the converse is true, namely that every $s - t$ flow is made up of $s - t$ paths and cycles.

Theorem 1.1.1 (Flow Decomposition Theorem)**.** *Any $s-t$ flow can be decomposed into a collection of $s-t$ paths and directed cycles.*

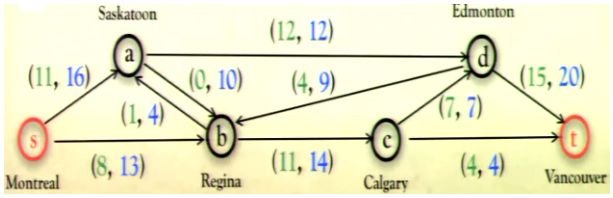
Proof. See section 5.1

□

Remark. The decomposition in the Flow Decomposition Theorem (FDT for short) need not be unique.

1.2 An Example - Trucking Syrup

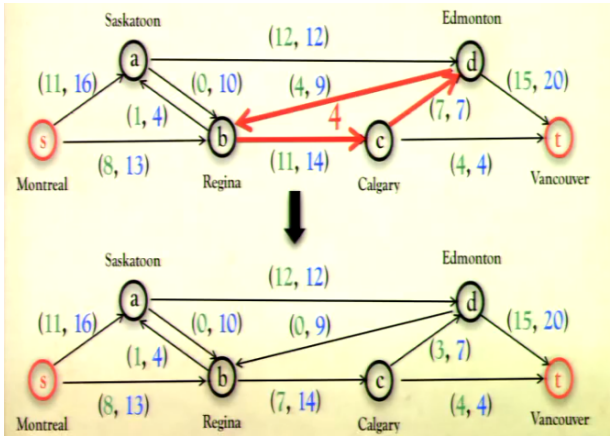
Suppose we send maple syrup on a network from Montreal to Vancouver. For each link a in the network, let u_a be the trucking capacity in tons on the link and f_a be the amount of syrup we are transporting on that link.



Note that we do indeed have an $s-t$ flow with flow value of 19 tons.

1.2.1 An Observation

If a flow puts positive weight on a directed cycle, we can remove this weight from the cycle and still have a flow of the same value.



So, for any $s-t$ flow, there is an $s-t$ flow of the same value that contains NO directed cycles. So combining this observation with theorem 5.1.1, we have that there is a max flow whose flow decomposition contains only directed paths.

2

Ford-Fulkerson Algorithm

2.1 Greedy Approach

Our first attempt at finding a max flow is the greedy approach.

Algorithm 2.1.1 (GreedyMaxFlow)**.**

1. Find a directed path P from s to t
2. Send as much flow on the path P as possible
3. Repeat until there is no surplus capacity on any $s - t$ path

This Greedy Algorithm does not work:

2.1.1 Example

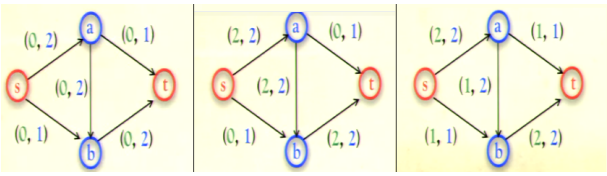
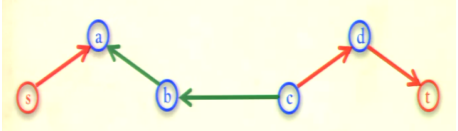


Figure 2.1: At the left is the start. The middle is the result of the Greedy Max Flow Algorithm. The right is a max flow.

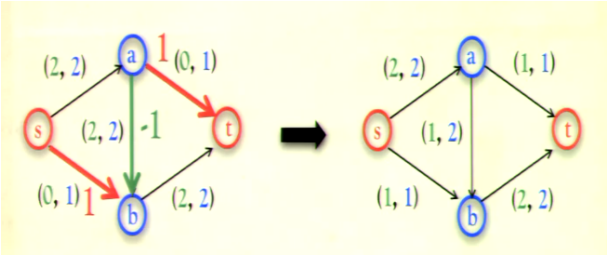
2.2 Augmentation Paths

Definition 2.2.1. An $s - t$ **augmenting path** is a path P from s to t where some of the arcs can be in the reverse direction.

Example 2.2.1.

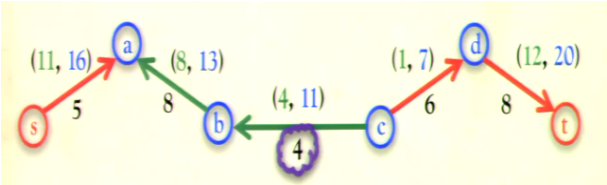


Suppose we increase the flow on a forward arc but decrease the flow on a backward arc on the path P. Then, we actually have that flow conservation is still maintained.



So, using a backwards arc corresponds to reducing the amount of flow used on that arc in the forward direction. So we are correcting a mistake we made earlier.

Now, we know that using an augmenting path has two nice properties. Namely, it maintains flow conservation and increases flow value. But can we ensure that the capacity constraints remain satisfied? In particular, can we guarantee that $f_a \geq 0$ isn't violated? Well, given the current flow $\vec{f}$, we can increase the flow on a forward arc (i, j) by $u_{ij} - f_{ij}$. And, we can decrease the flow on a backward arc (i, j) by f_{ij}.



In particular, we can increase the flow on the augmenting path P by the bottleneck capacity of P w.r.t flow $\vec{f}$:

$$b(P, \vec{f}) = \min \left[\min_{(i,j)\ forward\ arc} u_{ij} - f_{ij}, \min_{(i,j)\ backward\ arc} f_{ij} \right]$$

So, we have derived the Ford-Fulkerson Algorithm:

Algorithm 2.2.1 (Ford-Fulkerson)**.** Set $\vec{f} = 0$. Repeat the following:

1. Find an augmenting path P w.r.t $\vec{f}$
2. Augment flow on path P by $b(P, \vec{f})$

2.3 The Residual Graph

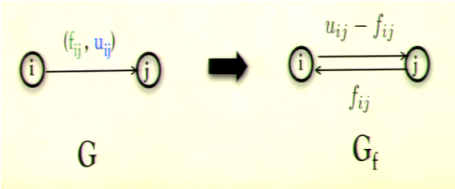
In this section, we examine the key to both implementing and understanding the Ford-Fulkerson Algorithm - residual graphs. With this tool, we will restate the Ford-Fulkerson algorithm.

Recall that in the Ford-Fulkerson algorithm , we had to use arcs in both the forward and backward directions. We are allowed to use arcs in the forward direction if the arc $a = (i, j)$ has spare capacity; $f_{ij} < u_{ij}$. We are allowed to use arcs in the backward direction if the arc $a = (i, j)$ has already been used forwards and thus flow can be sent back (so need $f_{ij} > 0$).

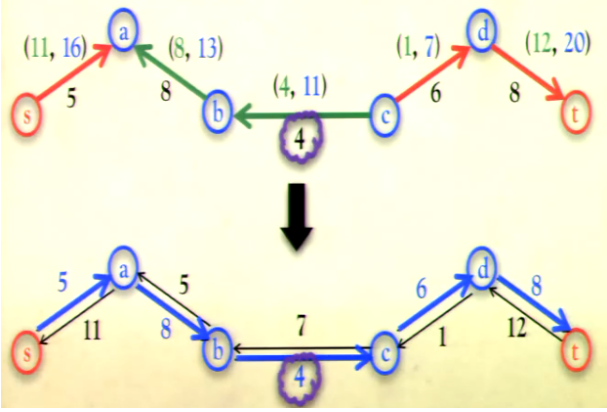
Definition 2.3.1. Given $G = (V, A)$ and a flow $\vec{f}$, we define the residual graph G_f as follows: For each arc $a = (i, j) \in A$

- G_f contains an arc (i, j) of capacity $u_a - f_a$
- G_f contains an arc (j, i) of capacity f_a

Example 2.3.1.



Observe that the arcs in the residual graph only have capacities. The bottleneck capacity of a path is now just the minimum capacity of an arc in the corresponding directed path in the residual graph.



So, we restate the Ford-Fulkerson Algorithm:

Algorithm 2.3.1 (Ford-Fulkerson v2)**.** Set $\vec{f} = 0$. Repeat the following:

1. Find a directed $s - t$ path P in the residual graph G_f
2. Augment flow on path P by its bottleneck capacity $b(P, \vec{f})$

Example 2.3.2. See lecture recording. Understand how this algorithm works to find a max flow!!!

Remark. We note that the Ford-Fulkerson algorithm terminates when there are no $s - t$ paths left in the residual graph.

2.4 Maxflow-Mincut Theorem

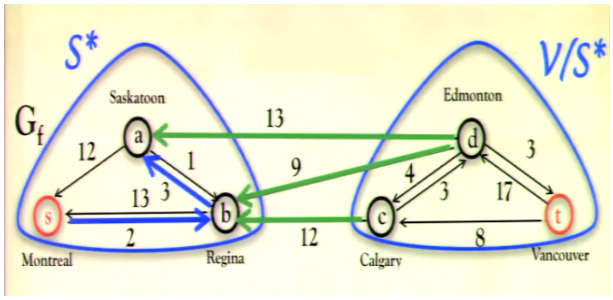
Given our newly updated Ford-Fulkerson algorithm, we would like to prove it works, finds a max flow, and we would like to examine its efficiency.

Definition 2.4.1. We say that $(\mathcal{S}, V \setminus \mathcal{S})$ is an $s - t$ cut if $s \in \mathcal{S}$ and $t \notin \mathcal{S}$

Let $\vec{f}^*$ be the flow output when the Ford-Fulkerson algorithm terminates. Let $\mathcal{S}^*$ denote the set of vertices reachable from s in the residual graph G_{f^*} i.e.

$$\mathcal{S}^* = \{v : \exists \text{ directed path from } s \text{ to } v \text{ in } G_{f^*}\}$$

Note that $(\mathcal{S}^*, V \setminus \mathcal{S}^*)$ is an $s - t$ cut (since trivially $s \in \mathcal{S}^*$ and by the termination condition, $t \notin \mathcal{S}^*$). Also, there are no arcs leaving $\mathcal{S}^*$ in G_{f^*} (if not then we could add an extra vertex to $\mathcal{S}^*$).



Lemma 2.4.1 (The Cut Lemma)**.** Given a flow $\vec{f}$ and an $s-t$ cut $(\mathcal{S}, V \setminus \mathcal{S})$, the value of the flow is

$$|f| = \sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a$$

Proof. The value of the flow is the amount of flow leaving the source s, so

$$\begin{aligned}
|f| &= \sum_{a \in \delta^+(s)} f_a \\
&= \sum_{a \in \delta^+(s)} - \sum_{a \in \delta^-(s)} f_a \text{ (the second sum we introduce is 0 because } \delta^-(s) = \emptyset) \\
&= \left(\sum_{a \in \delta^+(s)} - \sum_{a \in \delta^-(s)} f_a \right) + \sum_{v \in \mathcal{S} \setminus \{s\}} \left(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \right) \\
&= \sum_{v \in \mathcal{S}} \left(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a \right)
\end{aligned}$$

(where we have used that $(\sum_{a \in \delta^+(v)} f_a - \sum_{a \in \delta^-(v)} f_a)$=0 via flow conservation). Now, take any arc $a = (i, j) \in A$ in the graph. There are four cases we need to deal with.

1. If $(i, j) \in \delta^+(\mathcal{S}^*)$ then f_a appears once in the sum with coefficient 1
2. If $(i, j) \in \delta^-(\mathcal{S}^*)$ then f_a appears once in the sum with coefficient -1
3. If $\{i, j\} \subset V \setminus \mathcal{S}^*$ then f_a does not appear in the sum.
4. If $\{i, j\} \subset \mathcal{S}^*$ then f_a appears twice in the sum with coefficients -1 and 1. So , we have cancellation.

Therefore, $|f| = \sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a$ □

Definition 2.4.2. We define the capacity of an $s-t$ cut $(\mathcal{S}, V \setminus \mathcal{S})$ as

$$Cap(\mathcal{S}) = \sum_{a \in \delta^+(\mathcal{S})} u_a$$

Lemma 2.4.2. Given any flow $\vec{f}$ and any $s-t$ cut $(\mathcal{S}, V \setminus \mathcal{S})$ we have that

$$|f| \leq Cap(\mathcal{S})$$

Proof. Using the Cut lemma 2.4.1, we have

$$\begin{aligned}
|f| &= \sum_{a \in \delta^+(s)} f_a - \sum_{a \in \delta^-(s)} f_a \\
&\leq \sum_{a \in \delta^+(\mathcal{S})} u_a + 0
\end{aligned}$$

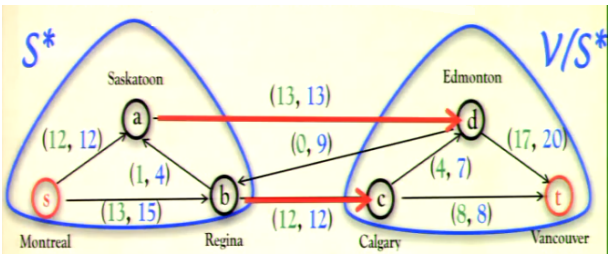
□

So, in particular, the value of a max flow is at most the capacity of the minimum cut (cut with the minimum capacity). In fact, they are equal:

Theorem 2.4.1 (Maxflow-Mincut)**.** *The max value of an $s-t$ flow is equal to the minimum capacity of an $s-t$ cut.*

Proof. By the above lemma 2.4.2, we have that $|f^*| \leq Cap(\mathcal{S}^*)$. Thus, we need only to show that $|f^*| \geq Cap(\mathcal{S}^*)$.

Now, recall that there are no arcs leaving $\mathcal{S}^*$ in G_{f^*}. What does this imply about the flow $\vec{f}^*$? Well, take any arc $(i,j) \in \delta^+(S^*)$ (this arc is not in the residual graph). Then ${f_{ij}}^* = u_{ij}$.



On the other hand, take any arc $(i,j) \in \delta^-(\mathcal{S}^*)$. This arc is not in the residual graph. Then, ${f_{ij}}^* = 0$.

So, by the Cut lemma 2.4.1, we have:

$$\begin{aligned} |f^*| &= \sum_{a \in \delta^+(\mathcal{S}^*)} f_a - \sum_{a \in \delta^-(\mathcal{S}^*)} f_a \\ &= \sum_{a \in \delta^+(\mathcal{S}^*)} u_a - \sum_{a \in \delta^-(\mathcal{S}^*)} f_a \\ &= \sum_{a \in \delta^+(\mathcal{S}^*)} u_a - 0 \\ &= Cap(\mathcal{S}^*) \end{aligned}$$

□

Immediately from the Maxflow-Mincut theorem 2.4.1, we have that the Ford-Fulkerson algorithm works and finds a max flow. Is it efficient though? Well, we can find G_f in $O(m)$ and we can find an $s-t$ path P in G_f in $O(m)$. Finally, we can augment the flow on P in $O(n)$. So, the run-time is $O(m \cdot N)$ where N is the number of iterations. To find N, recall that the algorithm terminates when $b(P, \vec{f}) = 0$ for every $s-t$ path P. Thus, since the capacities are integral, we have that $b(P, \vec{f}) \geq 1$ while running. So, in the worst case, we increase the flow value by only 1 each iteration. By the Maxflow-Mincut theorem, the max flow equals the min cut value C. If $U = \max_a u_a$ then $C \leq U$, so the run-time is

$$O(mn \cdot U)$$

which is only pseudo-polynomial. Recall that this is not truly polynomial time in input size. For

example, if the integers have b bits then U could be 2^b.

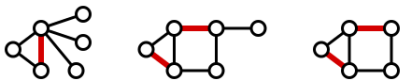
3 Maxflow Extensions

Remark. Out of this document, this chapter has the highest chance of typos/missing info because the lecture was not recorded and I am a bad student so I do not attend lectures. My apologies.

3.1 Max Matchings in Bipartite Graphs

Definition 3.1.1. A **matching** is a set of edges without common vertices.

Example 3.1.1.

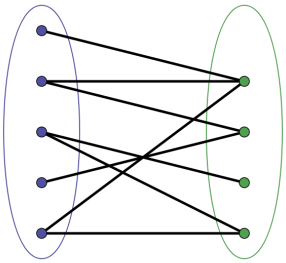


Definition 3.1.2. A **perfect matching** of a graph is a matching in which every vertex of the graph is incident to exactly one edge of the matching.

Definition 3.1.3. A **bipartite graph** is a graph whose vertices can be divided into two disjoint independent sets X, Y such that every edge connects a vertex in X to a vertex in Y. We write $V = X \cup Y$.

Remark. The independence of X, Y means that we cannot have an edge between two vertices in X (or Y).

Example 3.1.2.



The Maximum Matching problem for bipartite graphs is to find a maximum cardinality matching in a bipartite graph. We can do this efficiently using max flows.

3.1.1 Auxiliary Networks

An auxiliary network is constructed by:

1. Take an undirected bipartite graph $G = (X \cup Y, E)$
2. Direct each edge from x_i to y_i
3. Add source vertex s with outgoing arcs to each vertex in X
4. Add sink vertex t with incoming arcs from each vertex in Y
5. Give each arc a capacity of one.

We can now treat the auxiliary network like a flow. It will have value k since we have k distinct paths. Then, remove the source and sink and we have reduced the problem to that of a max flow.

3.1.2 Runtime

Recall that using the Ford-Fulkerson Algorithm we have a runtime of $O(m \cdot N)$ where N is the number of iterations. But what is the N here? Well, the max cardinality of a matching is at most $\min\{|X|, |Y|\} \leq n$. So, $N = n$ and thus the run-time is $O(mn)$, which is polynomial.

Now, recall from the Maxflow-Mincut Theorem that the max value of an $s - t$ flow is equal to the min. capacity of an $s - t$ cut. So, we can find a min cut using an auxiliary network in polynomial time.

Let give the arcs between X-vertices and Y-vertices infinite capacity. We note that this changes nothing since each edge into each X-vertex still has capacity one, so the max flow and min cut remain the same. Recall that the max flow has value at most n since s has n outgoing arcs. So, by the Maxflow-Mincut theorem, the capacity of the min cut $\mathcal{S}^*$ is at most n. Thus, the capacity of $\mathcal{S}^*$ is finite and thus $\delta^+(\mathcal{S}^*)$ contains no infinite capacity arcs.

Now, lets examine the structure of the mincut. Let $X^* = X \cap \mathcal{S}^*$. Then $\Gamma(X^*) \subset \mathcal{S}^* \cap Y$ otherwise $Cap(\mathcal{S}^*) = \sum_{a \in \delta^+(\mathcal{S}^*)} u_a = \infty$. So, all arcs in $\delta^+(\mathcal{S}^*)$ are of the form (s, x_i) and (y_j, t).

3.2 The Vertex Cover Problem

Consider an undirected graph $G = (V, E)$.

Definition 3.2.1. A **vertex cover** is a set $\mathcal{C} \subset V$ that touch every edge in the graph.

So the Vertex Cover Problem here is to find a min. cardinality vertex cover in a graph.

There are only 3 types of edges in a bipartite graph. Edges between X^* and $Y \setminus \Gamma(X^*)$ cannot occur, otherwise we would have infinite capacity. So, $(X \setminus X^*) \cup \Gamma(X^*)$ is a vertex cover. But, the cardinality of the vertex cover is the cardinality of the cut, thus this is a min cover.

3.2.1 Efficient Algorithms

When we run the max flow algorithm in an auxiliary network, we also find a min cut. The min cut gives us the min vertex cover in the original graph. So, there is a polynomial time algorithm to find the minimum cardinality vertex cover in a bipartite graph. If a vertex cover $\mathcal{C}$ and a matching M have the same cardinality, then they are both optimal (in a bipartite graph). I.e. $\mathcal{C}$ is a minimum vertex cover and M is a max matching.

3.3 Supply and Demand

3.3.1 Single Source/Sink

Suppose we have a source s that has a fixed supply amount equal to b. Suppose we have a sink t that has a demand of b (which is equivalent to a supply of $-b$). We would like to find the flow that satisfies the supply/demand constraint

$$flow - out(s) = flow - in(t) = b$$

This can be done iff there exists an $s-t$ flow of value at least b. So, all we need to do is run our max flow algorithm.

3.3.2 Multiple Sources/Sinks

Suppose we have sources $\{s_1, ..., s_k\}$ and sinks $\{t_1, ..., t_l\}$ where source s_i has supply b_i and sink t_j has demand b_j. Is there a flow that satisfies $flowout(v) - flowin(v) = b_v$ for all $v \in V$? ($b_v = 0$ for non source/sink vertices).

Well, we can convert this into a single source and single flow problem by

- Adding a super-source s and arcs (s, s_i) with capacity b_{s_i}
- Adding a super-sink t and arcs (t_j, t) with capacity b_{t_j}

There is a feasible flow to this multi-source problem iff there exists a flow from $s-t$ with value $\sum_{v:b_v>0} b_v$ (i.e. all the supply = all the demand).

4 A Few Applications of Maxflow

In this chapter, we examine three specific examples of applications of network flows/max flow problem.

4.1 Simplified Flight Scheduling

Suppose we have a collection of flights

Origin-Destination	Time
Boston - San Francisco	7am - 9am
Toronto - Boston	7am - 8am
Montreal - Toronto	8am - 9am
Toronto - San Francisco	10am - 1pm
Montreal - Toronto	10am - 11am
San Francisco - Montreal	10am - 5pm
San Francisco - Los Angeles	3pm - 4pm
Boston - Toronto	3pm - 4pm

How do we service all these flights using as few planes as possible? Well, we use network flows!

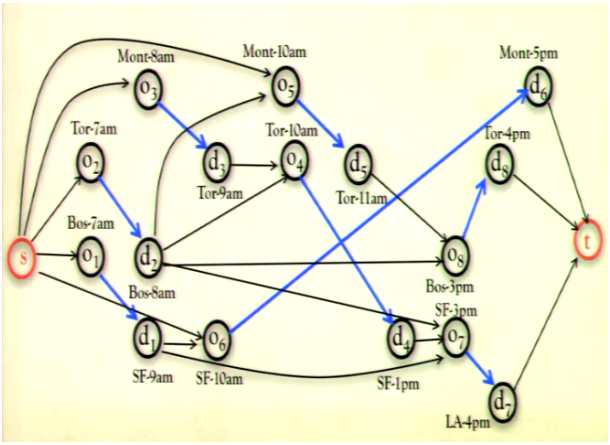
4.1.1 Network Flow Model

We create the model as follows:

- For each flight i, we have a vertex for the origin o_i and the destination d_i
- Add a source vertex s with supply k (k is the number of planes)
- Add arc (s, o_i) with capacity one for each flight i

- Add a sink vertex t with demand k
- Add an arc (d_i, t) with capacity 1 for every flight i
- Add an arc (d_i, o_j) with capacity one if it is feasible for a plane to service flight i and then service flight j
- Finally, we have an arc (o_i, d_i) with a lower bound of one.

Example 4.1.1.



Remark. We note that our model is an oversimplification. In reality, airline scheduling is rather complex; need to take into account costs, crew constraints, plane capacities, multiple days, etc. We could actually incorporate these into a network flow model, but we will not do so in the class.

4.2 Open Pit Mining

Suppose we have a set V of blocks

-20	-35	-10	-10	-20	0	-35	-10	-20	-15
10	15	-10	15	0	5	-10	30	90	-5
-5	20	-25	70	50	0	-30	10	-20	-15
0	5	0	0	-60	5	-90	-30	5	10
20	15	-10	15	80	50	-10	30	0	-5
0	10	-25	-10	10	20	-20	15	0	15

Each block $i \in V$ has associated profit π_i (the profit is the estimated value of the ore contained in the block minus the cost of digging up and processing the block).

We want to maximize the profit of the pit. Initially, this sounds easy, but there are some topological constraints that we must consider. We require that the pit cannot be too steep. In particular, to dig a block i, we must first remove the three closest blocks in the layer above.

Example 4.2.1.

-20	-35	-10	-10	-20	0	-35	-10	-20	-15
10	15	-10	15	0	5	-10	30	90	-5
-5	20	-25	70	50	0	-30	10	-20	-15
0	5	0	0	-60	5	-90	-30	5	10
20	15	-10	15	80	50	-10	30	0	-5
0	10	-25	-10	10	20	-20	15	0	15

Figure 4.1: To get to the blue block, here we would have to dig all the red ones also

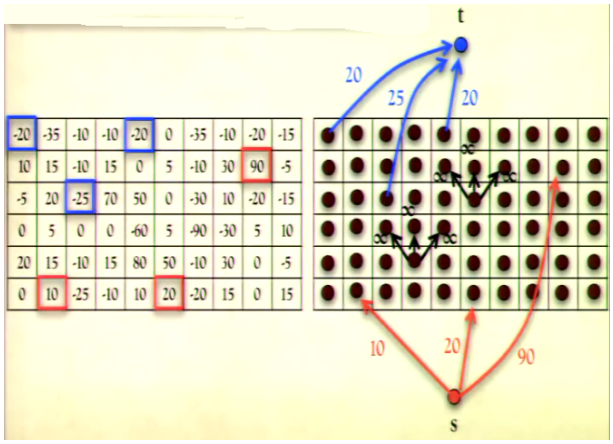
To solve this problem, we use Network flows

4.2.1 Network Flow Model

We create the model as follows:

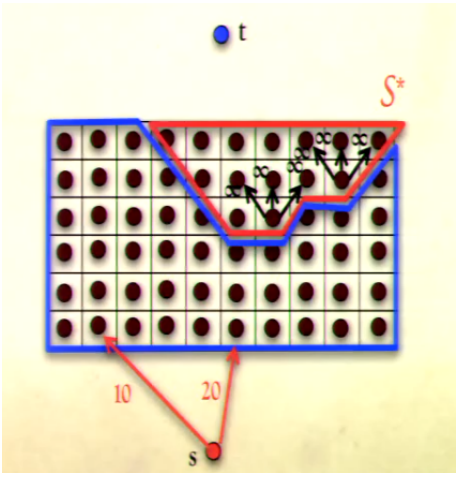
- There is a vertex for each block $i \in V$
- There is a source vertex s and sink vertex t
- There is an arc (s, i) of capacity π_i iff $\pi_i > 0$
- There is an arc (j, t) of capacity $|\pi_i|$ iff $\pi_i < 0$
- There is an arc from i to each of the 3 blocks above it of capacity ∞.

Example 4.2.2.



Observe that $Cap(\{s\}) = \sum_{i:\pi_i>0} \pi_i$ which is finite. So, the capacity of the min cut $\mathcal{S}^*$ is finite and therefore by the Maxflow-Mincut theorem 2.4.1, the max flow value is finite.

Now, suppose the min cut $\mathcal{S}^*$ contains a block i. Then, the 3 blocks above i are in $\mathcal{S}^*$ since the corresponding arcs have infinite capacity. So then the three blocks above each of the previous three block are also in $\mathcal{S}^*$ etc... Thus, $\mathcal{S}^*$ is a feasible pit.



4.2.2 Capacity of a Cut

What exactly is the capacity of an $s-t$ cut $\mathcal{S}$? Well,

$$\begin{aligned}
Cap(\mathcal{S}) &= \sum_{a\in\delta^+(\mathcal{S})} u_a \\
&= \sum_{i\notin\mathcal{S}:\pi_i>0} \pi_i + \sum_{i\in\mathcal{S}:\pi_i<0} |\pi_i| \\
&= \left(\sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}:\pi_i>0} \pi_i\right) + \sum_{i\in\mathcal{S}:\pi_i<0} |\pi_i| \\
&= \left(\sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}:\pi_i>0} \pi_i\right) - \sum_{i\in\mathcal{S}:\pi_i<0} \pi_i \\
&= \sum_{i\in V:\pi_i>0} \pi_i - \sum_{i\in\mathcal{S}} \pi_i \\
&= \Phi - \sum_{i\in\mathcal{S}} \pi_i
\end{aligned}$$

Observe that Φ is a constant and is independent of the cut $\mathcal{S}$. Thus, the capacity of $\mathcal{S}$ is maximized exactly when the profit associated with $\mathcal{S}$ is minimized. So, the min cut $\mathcal{S}^*$ gives an optimum pit design. (We can find the min cut by running the max flow algorithm).

4.3 Image Segmentation

A basic task is foreground/background segmentation. So, lets label the pixels according to whether they belong to the foreground or the background.

4.3.1 Network Flow Model

Our model has two parameters for each pixel:

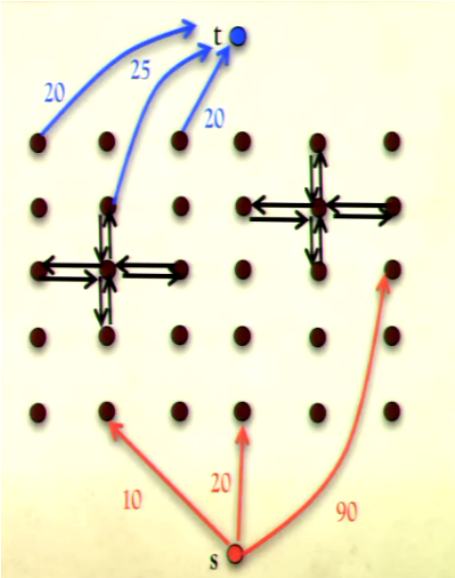
- Let f_i be the likelihood that pixel i is in the foreground
- Let b_i be the likelihood that pixel i is in the background

In general, if $f_i > b_i$ then we want pixel i in the foreground. But, of course we also want a smooth boundary between the foreground and background. To incorporate this, we introduce a penalty ρ_{ij} for separating adjacent pixels i and j. Now, we create the model as follows:

- There is a vertex for each pixel i

- There is a source (foreground) vertex s and sink (background) vertex t
- There is an arc (s, i) of capacity f_i
- There is an arc (j, t) of capacity b_j
- For adjacent pixels, there are arcs (i, j) and (j, i) both of capacity ρ_{ij}

Example 4.3.1.



4.3.2 Capacity of a Cut

What exactly is the capacity of an $s-t$ cut $\mathcal{S}$? Well,

$$\begin{aligned}
Cap(\mathcal{S}) &= \sum_{a \in \delta^+(\mathcal{S})} u_a \\
&= \sum_{i \notin \mathcal{S}} f_i + \sum_{j \in \mathcal{S}} b_j + \sum_{(i,j) \in \delta^+(\mathcal{S})} \rho_{ij} \\
&= (\sum_{i \in V} f_i - \sum_{i \in \mathcal{S}} f_i) + (\sum_{j \in V} b_j - \sum_{j \notin \mathcal{S}} b_j) + \sum_{(i,j) \in \delta^+(\mathcal{S})} \rho_{ij} \\
&= \sum_{i \in V} (f_i + b_i) - \sum_{i \in \mathcal{S}} f_i - \sum_{j \notin \mathcal{S}} b_j + \sum_{(i,j) \in \delta^+(\mathcal{S})} \rho_{ij} \\
&= \Phi - (\sum_{i \in \mathcal{S}} f_i + \sum_{j \notin \mathcal{S}} b_j - \sum_{(i,j) \in \delta^+(\mathcal{S})} \rho_{ij})
\end{aligned}$$

Observe that Φ is a constant that is independent of the cut $\mathcal{S}$. So, the capacity of $\mathcal{S}$ is minimized when the bracketed term is maximized. But, this is exactly the segmentation problem

$$\max_{\mathcal{S}} \sum_{i \in \mathcal{S}} f_i + \sum_{j \notin \mathcal{S}} b_j - \sum_{(i,j) \in \delta^+(\mathcal{S})} \rho_{ij}$$

So, we can solve the segmentation problem by finding a min cut in the graph.

5 The Maximum Capacity Augmenting Path Algorithm

In this chapter we address the issue that we had with the Ford-Fulkerson Algorithm; namely that our answer to its efficiency wasn't really satisfactory. Recall that the Ford-Fulkerson Algorithm has a run-time of $O(m \cdot N)$ where N is the number of iterations, and could be $n \cdot U$ where $U = \max_{a \in A} u_a$. This of course is pseudo-polynomial time. So, the purpose of this chapter is to see if we can do better. Using (and finally proving) the Flow Decomposition Theorem, we find that by choosing the paths in Ford-Fulkerson more carefully, we arrive at a (weakly) polynomial time algorithm for finding a Max Flow; this algorithm is called the Maximum Capacity Augmenting Path Algorithm (or MCAPalg for short).

5.1 Proving the Flow Decomposition Theorem

Recall the Flow Decomposition Theorem:

Theorem 5.1.1 (Flow Decomposition Theorem)**.** *Any $s-t$ flow can be decomposed into a collection of $s-t$ paths and directed cycles of cardinality at most m where m is the number of arcs.*

(here we've stated a slightly stronger version)

To prove this theorem, we need one lemma:

Lemma 5.1.1. Any $s-t$ flow $\vec{f} \neq \vec{0}$ contains a path or cycle.

Proof. If $\vec{f}$ contains a directed cycle C, we are done. So, assume that $\vec{f}$ contains no directed cycles. Now, we just need to prove that we have a path.

Note that since $\vec{f}$ has no cycles and $\vec{f} \neq \vec{0}$, $\vec{f}$ has a flow value of at least one. Thus $\sum_{a \in \delta^+(\{s\})} f_a \geq 1$. So in particular, $\exists$ at least one arc $a_1 = (s, v_1)$ with $f_{a_1} \geq 1$.

But then, by flow conservation, there must be an arc in $\vec{f}$ leaving v_1. So, since $\vec{f}$ contains no cycles, this arc $a_2 = (v_1, v_2)$ goes to a new vertex v_2. But then again by flow conservation there is an arc in $\vec{f}$ leaving v_2. Keep repeating. Since the graph is finite, this process must terminate at the sink vertex t. So, we have an $s-t$ path P with $\min_{a \in P} f_a \geq 1$. □

Now that we have our lemma, we can prove the Flow Decomposition Theorem.

Proof. We proceed via induction on the number of arcs m in the graph.

For the base cases: Note that if $m = 0$ then $\vec{f}$ is empty and trivially consist of zero paths and cycles. If $m = 1$ then $\vec{f}$ is simply an arc (s, t) so that $\vec{f}$ can be decomposed into one $s-t$ path.

Now, assume that any flow with $k < m$ arcs can be decomposed into a collection of at most k cycles and $s-t$ paths.

Take any flow $\vec{f}$ with m arcs. By our lemma 5.1.1, $\vec{f}$ contains a path or cycle.

The first case to deal with is if $\vec{f}$ contains a cycle C. Then, let $\vec{\hat{f}}$ be the flow obtained by removing $\min_{a\in C} f_a$ units of flow from each arc in the cycle C. Then, $\vec{\hat{f}}$ has at least one fewer arc than $\vec{f}$. So, by hypothesis, $\vec{\hat{f}}$ can be decomposed into at most $m-1$ paths and cycles. Thus, $\vec{f}$ can be decomposed into at most m paths and cycles.

The second case to deal with is if $\vec{f}$ contains a path P. Then, let $\vec{\hat{f}}$ be the flow obtained by removing $\min_{a\in P} f_a$ units of flow from each arc in the path P. Then, $\vec{\hat{f}}$ has at least one fewer arc than $\vec{f}$. So by hypothesis, $\vec{\hat{f}}$ can be decomposed into at most $m-1$ paths and cycles. Thus, $\vec{f}$ can be decomposed into at most m paths and cycles.

□

5.2 Choosing the Paths

So, the Flow Decomposition theorem 5.1.1 tells us that the max flow can be decomposed into at most m paths and cycles. However, since cycles do not increase the flow value, there is a max flow $\vec{f^*}$ that decomposes into at most m paths. So, suppose the Ford-Fulkerson Algorithm chooses the paths in this decomposition. Then, the run-time would be $O(m^2)$. But, we don't know what $\vec{f^*}$ is (its what the algorithm finds!), so we don't know its flow decomposition. Even though we cant find this exact set of paths, maybe we can find a different set of paths that is almost as good. Perhaps the most natural approach is to select the paths greedily:

Algorithm 5.2.1 (MCAPalg)**.** Set $\vec{f} = \vec{0}$. Repeat the following:

1. Find a maximum capacity augmenting path P w.r.t $\vec{f}$
2. Augment flow on path P by $b(P, \vec{f})$

(So, we are choosing the path that increases the flow value by the largest amount possible at that specific iteration.)

5.3 Runtime of MCAPalg

The purpose of this section is to prove the following:

Theorem 5.3.1 (Runtime of MCAPalg)**.** *The MCAPalg runs in time* $O(m^3(\ln n + \ln U))$.

Lemma 5.3.1. Let $\vec{f^*}$ be a max flow. Then, there is a path P in G with capacity at least $\frac{1}{m}|\vec{f^*}|$

Proof. By the Flow Decomposition theorem 5.1.1, $\vec{f^*}$ consists of at most m paths. Therefore, at least one of these paths carries a one m-th fraction of the total flow value □

Lemma 5.3.2. Let $\vec{f}$ be a flow and let $\vec{f^*}$ be a max flow. Then, in the residual graph G_f, there is a path P with

$$b(P, \vec{f}) \geq \frac{|\vec{f^*}| - |\vec{f}|}{m}$$

Proof. Note that $(\vec{f^*} - \vec{f})$ satisfies flow conservation. So, $(\vec{f^*} - \vec{f})$ can be decomposed into at most m paths and cycles. We may assume that $(\vec{f^*} - \vec{f})$ can be decomposed into at most m paths. Therefore, at least one of these paths carries a one m-th fraction of the difference in the flow values □

Lemma 5.3.3. The MCAPalg terminates in at most $m(\ln n + \ln U)$ iterations where $U = \max_{a\in A} u_a$.

Proof. Let the algorithm find paths $\{P_1, ..., P_T\}$. We need to show that $T \leq m(\ln n + \ln U)$.

Let $\vec{f}_t$ be the flow found after t iterations. Previous lemma 5.3.2 tells us that path P_{t+1} satisfies

$$\begin{aligned} b(P_{t+1}, \vec{f}_t) &\geq \frac{|\vec{f^*}| - |\vec{f}_t|}{m} \\ &:= \frac{\Delta_{t+1}}{m} \end{aligned}$$

Here Δ_{t+1} is the flow quantity still to be found at the start of step $t+1$. Observe that $\Delta_1 = |\vec{f^*}| - |\vec{f}_0| = |\vec{f^*}| - |\vec{0}| = |\vec{f^*}|$

So, $b(P_{t+1}, \vec{f}_t) \geq \frac{\Delta_t}{m}$ and thus

$$\begin{aligned} \Delta_{t+1} &\leq \Delta_t - \frac{1}{m}\Delta_t \\ &= (1 - \frac{1}{m})\Delta_t \\ &\leq (1 - \frac{1}{m})(1 - \frac{1}{m})\Delta_{t-1} \\ &\leq \cdots \\ &\leq (1 - \frac{1}{m})^t \Delta_1 \\ &= (1 - \frac{1}{m})^t |\vec{f^*}| \\ &< (e^{\frac{-1}{m}})^t |\vec{f^*}| \\ &= e^{\frac{-t}{m}} |\vec{f^*}| \end{aligned}$$

So, setting $t = m \ln |\vec{f^*}|$ we get that $\Delta_{t+1} < 1$. So after $T = m \ln |\vec{f^*}|$ steps, the quantity of flow remaining to be found is < 1. So we have found a max flow.

Now, the max flow has a value at most $n \cdot U$, so

$$\ln |\vec{f^*}| \leq \ln(n \cdot U) = \ln n + \ln U$$

So, $T \leq m(\ln n + \ln U)$ □

Lemma 5.3.4. The MCAPalg takes $O(m^2)$ time per iteration.

Proof. It remains to calculate how long it takes to find the max capacity augmenting path. Label the arcs $\{1, 2, ..., 2m\}$ in the residual graph in decreasing order of residual capacity.

We can test if there exists an $s - t$ path using only arcs in $\{1, ..., k\}$ in $O(m)$ time. Then we can do this for all k in time $O(m^2)$.

The maximum capacity augmenting path is the path we find using the smallest k for which an $s - t$ path exists using only arcs in $\{1, ..., k\}$ □

So, putting lemma 5.3.3 and lemma 5.3.4 together, we have proven theorem 5.3.1.

Acknowledgements

These notes have been transcribed from the lectures given by Prof. Adrian Vetta. Any figures are screenshots of the recorded lectures, or taken from Wikepedia. This document is for personal use only, and beware of typos!